Core Java
Functional Programming

Akshita Chanchlani

Functional Programming

Package : java.util.function

Functional Programming (FP) is one of the type of programming pattern that helps the process of building application by using of higher order functions, avoiding shared state, mutable data.
Functional programming is the way of writing s/w applications that uses only pure fuctions & immutable values.

Need of Functional Programming

1. To write more readable , maintainable , clean & concise code.
2. To use APIs easily n effectively.
3. To enable parallel processing

Declarative vs Imperative :

- Functional programming is a declarative pattern means that the program logic is expressed without explicitly describing the flow control.(you will just have to specify what's to be done)

- Imperative programs spend lines of code describing the specific steps used to achieve the desired results by following flow control. (you will have to specify what's to be done & how)

- Declarative programs remove the flow control process, and instead spend lines of code describing the data flow.

Functional and Structure programming

- Structured programming gives importance to logical structure or process, whereas Functional programming mainly focuses on data.
- Structured programming divided small units or functions, whereas Functional programming divided into small run-time entities called objects.
- Structured programming less secure, whereas Functional programming highly secure.
- Structured programming can’t handle complex problems, whereas Functional programming handles any level of a complex problem.

Functions as Objects

- store in a variable, pass as a parameter, return as a result.
- Only variables and objects can be stored, passed, and returned.

- Functions can be represented as objects in Java. That is called as Functional Interface.

- We implement classes from Functional Interfaces.

- Then, we can create objects.

- These objects are called **First-Class Functions. Because** It is said that it is because functions are finally being treated as *first-class citizens*. When encapsulated in an object, they can finally be stored, returned, and used as parameters to other functions.

Functions as Expressions

Functions can be expressions; therefore, they can exist inside methods, as the expressions that we create combining operators and values

We will need a new kind of expression called **Lambda Expressions**. They are functions too.

Types of Functional Programming

- Streams functional Programming
- Lambda Expressions functional Programming
- Method Reference functional Programming

Streams Functional Programming

```
objectName.stream();
```

Lambda Expressions

- Lambda expression used to represent a method interface with an expression.
- It helps to iterate, filtering and extracting data from collections.
- Lambda expression interface implementation is a functional interface.
- It reduces a lot of code.
- Lambda expression treated as a function so java compiler can't create .class

Syntax:

```
(arguments) ->
{
//code for implementation
}
Arguments:  argument-list can be have values or no values
Example: arguments1, arguments2, arguments3,……
->: Joins code implementation and arguments.
```

Lambda expression with a single argument

Syntax:

```
(argument1) ->
{
//code for implementation
}
```

Lambda expression without argument

Syntax:

```
() ->
{
//code for implementation
}
```

Method Reference

- Method reference used to refer to a method of functional interface.
- It is one more easy form of a lambda expression.
- If you use every time lambda expression to point a method, we can use method reference in place of method reference.

Syntax:

```
Class-Name:: static method name
```

Functional Programming Concepts

•**Higher-order functions:**

In functional programming, functions are to be considered as first-class citizens. That is, so far in the legacy style of coding, we can do below stuff with objects.

- We can pass **objects** to a function.
- We can create **objects** within function.
- We can return **objects** from a function.
- We can pass a **function** to a function.
- We can create a **function** within function.
- We can return a **function** from a function.

•**Pure functions:**

A function is called pure function if it always returns the same result for same argument values and it has no side effects like modifying an argument (or global variable) or outputting something.

•**Lambda expressions**

A Lambda expression is an anonymous method that has mutability at very minimum and it has only a parameter list and a body. The return type is always inferred based on the context. Also, make a note, Lambda expressions work in parallel with the functional interface.

The syntax of a lambda expression , `(parameter) -> body`

First-Class and Higher-Order Functions

- A programming language is said to have first-class functions if it treats functions as first-class citizens. Basically, it means that **functions are allowed to support all operations typically available to other entities**.
- These include assigning functions to variables, passing them as arguments to other functions, and returning them as values from other functions.
- This property makes it possible to define higher-order functions in functional programming. **Higher-order functions are capable of receiving function as arguments and returning a function as a result**.
- This further enables several techniques in functional programming like function composition and currying.
- Traditionally it was only possible to pass functions in Java using constructs like functional interfaces or anonymous inner classes. Functional interfaces have exactly one abstract method and are also known as Single Abstract Method (SAM) interfaces.

For example we have to provide a custom comparator to *Collections.sort* method:

```
Collections.sort(numbers, new Comparator<Integer>() {
@Override public int compare(Integer n1, Integer n2) {
return n1.compareTo(n2); } });
```

Functional Programming Functions

Functions are treated as a first class citizens, because You can

- define anonymous functions
- assign a function to a variable (function literal)
- pass function as a parameter
- return function as a return value
- We can pass **objects** to a function.
- We can create **objects** within function.
- We can return **objects** from a function.
- We can pass a **function** to a function.
- We can create a **function** within function.
- We can return a **function** from a function.

Functional Interface

- An interface which has exactly single abstract method(SAM) is called functional interface.

Eg. Runnable,Comparable,Comparator,Iterable etc.

- Annotation for Functional Interface

@FunctionalInterface

- Functional i/f references can be substituted by lambda expressions, method references, or constructor references.

Lambda Expression

- Lambdas are the most important new addition
- **Java treats a lambda expression as an *Object***, which is, in fact, the true first-class citizen in Java.
- A big challenge was to introduce lambdas without requiring recompilation of existing binaries

For example above example with Lambda Expression:

```
Collections.sort(numbers, (n1, n2) -> n1.compareTo(n2));
```

this is more concise and understandable

Syntax of Java 8 Lambdas

- A Java 8 lambda is basically a method in Java without a declaration usually written as (parameters) -> { body }. Examples,

```
1.  (int x, int y) -> { return x + y; }
2.  x -> x * x
3.  ( ) -> x
```

- A lambda can have zero or more parameters separated by commas and their type can be explicitly declared or inferred from the context.
- Parenthesis are not needed around a single parameter.
- ( ) is used to denote zero parameters.
- The body can contain zero or more statements.
- Braces are not needed around a single-statement body.

Benefits of Lambdas in Java 8

- Enabling functional programming
- Writing leaner more compact code
- Facilitating parallel programming
- Developing more generic, flexible and reusable APIs
- Being able to pass behaviors as well as data to functions

Example 1:
Print a list of integers with a lambda

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> System.out.println(x));
```

x -> System.out.println(x) is a lambda expression that defines an anonymous function with one parameter named x of type Integer

Example 2:
A multiline lambda

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> {
    x += 2;
    System.out.println(x);
});
```

Braces are needed to enclose a multiline body in a lambda expression.

Example 3:
A lambda with a defined local variable

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> {
   int y = x * 2;
   System.out.println(y);
});
```

- Just as with ordinary functions, you can define local variables inside the body of a lambda expression

Example 4:
A lambda with a declared parameter type

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach((Integer x ->) {
   x += 2;
   System.out.println(x);
});
```

- You can, if you wish, specify the parameter type.

Variable Capture

- Lambdas can interact with variables defined outside the body of the lambda

- Using these variables is called variable capture

Local Variable Capture Example

```
List<Integer> intSeq = Arrays.asList(1,2,3);

    int var = 10;
    intSeq.forEach(x -> System.out.println(x + var));
```

- Note: local variables used inside the body of a lambda must be final or effectively final

Static Variable Capture Example

```
public class Demo {
  private static int var = 10;
  public static void main(String[] args) {
    List<Integer> intSeq = Arrays.asList(1,2,3);
    intSeq.forEach(x -> System.out.println(x + var));
  }
}
```

Pure Functions

- **a pure function should return a value based only on the arguments and should have no side effects.**

Example find the sum of all the numbers we've just sorted

```
Integer sum(List<Integer> numbers) { return
numbers.stream().collect(Collectors.summingInt(Integer::intValue)); }
```

this method depends only on the arguments it receives, hence, it's deterministic. Moreover, it doesn't produce any side effects.

Side effects can be anything apart from the intended behavior of the method. For instance, **side-effects can be as simple as updating a local or global state** or saving to a database before returning a value.

Method References

- Method references can be used to pass an existing function in places where a lambda is expected
- The signature of the referenced method needs to match the signature of the functional interface method

Summary of Method References

Method Reference Type	Syntax	Example
static	ClassName::StaticMethodName	String::valueOf
constructor	ClassName::new	ArrayList::new
specific object instance	objectReference::MethodName	x::toString
arbitrary object of a given type	ClassName::InstanceMethodName	Object::toString

Conciseness with Method References

We can rewrite the statement

```
intSeq.forEach(x -> System.out.println(x));
```

more concisely using a method reference

```
intSeq.forEach(System.out::println);
```

What is a stream?

A stream is “a sequence of elements from a source that supports data processing operations.” Internal iteration.

Collections are held in memory space. You need the entire collection to iterate. SPACE

Streams are created on-demand and they are infinite. TIME

Declarative— More concise and readable
Composable— Greater flexibility
Parallelizable— Better performance

Stream API

- The new java.util.stream package provides utilities to support functional-style operations on streams of values.
- A common way to obtain a stream is from a collection:

```
Stream<T> stream = collection.stream();
```

- Streams can be sequential or parallel.
- Streams are useful for selecting values and performing actions on the results.

Sequence of Elements

Sequence of elements— Like a collection, a stream provides an interface to a sequenced set of values of a specific element type. Because collections are data structures, they're mostly about storing and accessing elements, but streams are **also** about expressing computations such as filter, sorted, and map.

Source

Source— Streams consume from a data-providing source such as collections, arrays, or I/O resources.

Note that generating a stream from an ordered collection preserves the ordering. The elements of a stream coming from a list will have the same order as the list.

Data processing

Data processing— Streams support database-like operations and common operations from functional programming languages to manipulate data, such as filter, map, reduce, find, match, sort, and so on. Stream operations can be executed either **sequentially** or in **parallel**.

Pipelining

Pipelining— Many stream operations return a stream themselves, allowing operations to be chained and form a larger pipeline. This enables certain optimizations such as laziness and short-circuiting. A pipeline of operations can be viewed as a database-like query on the data source.

Stream Operations

- An intermediate operation keeps a stream open for further operations. Intermediate operations are lazy.
- A terminal operation must be the final operation on a stream. Once a terminal operation is invoked, the stream is consumed and is no longer usable.

Example Intermediate Operations

- `filter` excludes all elements that don't match a Predicate.

- `map` performs a one-to-one transformation of elements using a Function.

A Stream Pipeline

A stream pipeline has three components:

1. A source such as a Collection, an array, a generator function, or an IO channel;
2. Zero or more intermediate operations; and
3. A terminal operation

Stream Example

```
int sum = widgets.stream()
                 .filter(w -> w.getColor() == RED)
                 .mapToInt(w -> w.getWeight())
                 .sum();
```

Here, `widgets` is a `Collection<Widget>`. We create a stream of `Widget` objects via `Collection.stream()`, filter it to produce a stream containing only the red widgets, and then transform it into a stream of `int` values representing the weight of each red widget. Then this stream is summed to produce a total weight.

Parting Example: Using lambdas and stream to sum the squares of the elements on a list

```
List<Integer> list = Arrays.asList(1,2,3);

int sum = list.stream().map(x -> x*x).reduce((x,y) -> x + y).get();

System.out.println(sum);
```

- Here map(x -> x*x) squares each element and then reduce((x,y) -> x + y) reduces all elements into a single number

http://viralpatel.net/blogs/lambda-expressions-java-tutorial/

Thank You.